Model-Based Testing and Validation of Control Software with Reactis®

November 2003

Reactive Systems, inc
www.reactive-systems.com

Contents

Reactive Systems, Incorporated
120-B East Broad Street
Falls Church, Virginia 22046
USA
www.reactive-systems.com
info@reactive-systems.com

1. Introduction

Model-based development (MBD) is growing in popularity among engineers who develop embedded control systems [Swo98, Bec00]. In MBD, executable visual models of embedded control systems are developed in advance of system implementation. The models may be used to drive the development of control software, and may also serve as a basis aoftware and system testing. One benefit of MBD is that it allows engineers to begin debugging and validation activities at design time, when the cost of detecting and dealing with design defects is much smaller than at the software and system implementation level. Another is that models may be used as a baseline for assessing implementation behavior during system testing and validation. For these reasons, judicious use of modeling can lead to quite dramatic over-all reductions in the cost control-system development, especially when robust tool support is available.

The Reactis® tool suite of Reactive Systems, Inc., substantially enhances the gains organizations realize from MBD by automating testing and validation processes based on Simulink® / Stateflow® models of control systems.[1] Using Reactis, engineers may:

- generate test suites from a model that thoroughly exercise the model; and
- determine whether or not a model can violate requirements on expected system behavior.

The tool also includes an array of sophisticated model-debugging features.

In this paper, we discuss how Reactis may be used to automate different development activities. In particular, we show how the tool may be used to develop more robust models, how it can streamline software and system testing, and how it may be used to support the reverse-engineering of models from existing control software.

2. An Overview of Reactis

A MBD environment involving Reactis, Simulink and Stateflow is depicted in Figure 1. Reactis contains three core components: Tester, which automatically generates test suites from models; Simulator, which enables users to visualize model execution; and Validator, which automatically searches models for violations of user-specified requirements. The remainder of this section desribes these components in more detail.

2.1. Reactis Tester

Reactis Tester automatically generates test suites from models. The test suites provide comprehensive coverage of different test-quality metrics—including the *Modified Condition/Decision Coverage* (MC/DC) test coverage measure mandated by the US Federal Aviation Administration (FAA) in its DO-178/B guidelines—while at the same time minimizing redundancy in tests. Each test case in a test suite consists of a sequence of inputs fed into the model as well as the responses

[1] MATLAB®, Simulink® and Stateflow® are registered trademarks of The MathWorks, Inc.

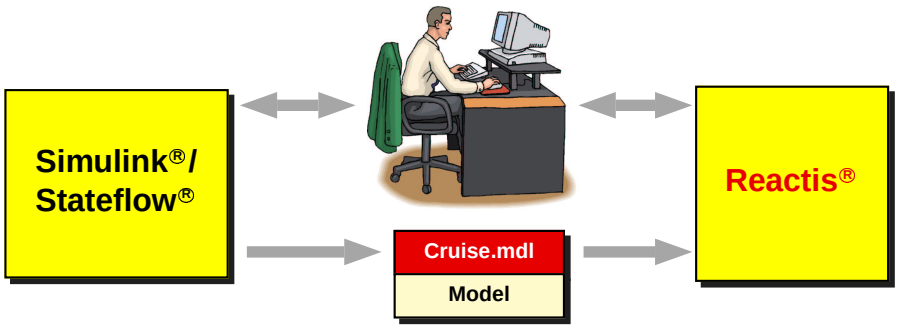

Figure 1: Reactis is packaged as a plug-in for use with the Simulink / Stateflow modeling environment.

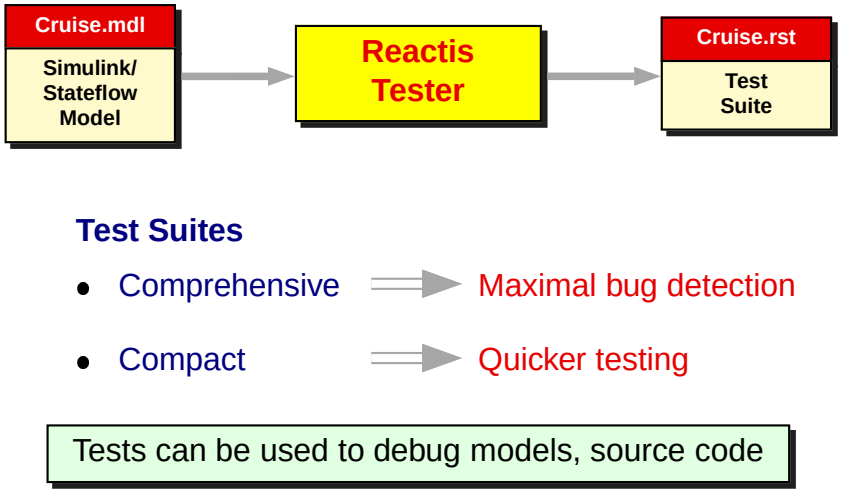

Figure 2: Reactis Tester automatically generates comprehensive yet compact test suites.

to those inputs generated by the model. These tests may then be used for a variety of purposes, including the following.

Implementation conformance. The tests may be applied to source-code implementations of models to ensure conformance with model behavior.

Model testing and debugging. The tests may be run on the models themselves to study and revise model behavior.

Reverse engineering of models from source. Tests may be generated from models derived from legacy code in order to check conformance between model and code.

Reactis Tester enables engineers to maximize the effectiveness of testing while reducing the time actually spent on testing.

The structure of Tester-generated test suites is shown in Figure 3. A test may be viewed as a matrix in which each row corresponds to either an inport or outport and each column represents a simulation step. Test suites are constructed by simulating a model and recording the input and output values at each step. The model computes the outputs at each step, but several approaches are possible for selecting the input values to drive simulation. The input data could be captured during field testing or constructed manually by an engineer, but these are expensive tasks. Alternatively, the inputs could be generated randomly; however, this approach yields tests with poor coverage.

Reactis Tester employs a novel approach called *guided simulation* to generate quality input data automatically. The idea behind this approach is to use algorithms and heuristics to automatically generate inputs that cause *coverage targets* (i.e. model elements that the user wants to ensure are executed at least once) that have not yet been covered to be executed. Reactis currently allows users to track several different classes of targets, or *coverage criteria*. Two of the criteria supported by Reactis involve only Simulink, three are specific to Stateflow, and the remaining are generic in the sense that they include targets within both the Simulink and the Stateflow portions of a model.

Simulink-specific: *Conditional subsystems*. *Branches* of the following blocks: Dead Zone, Logical Operator, MinMax, Multiport Switch, Relational Operator, Saturation, Switch.

Stateflow-specific *States. Condition actions. Transition actions.*

Generic *Decisions* from logic blocks in Simulink or transition segments in Stateflow. *Conditions* (the atomic predicates that are the building blocks of decisions). *Modified Condition/Decision Coverage* (MC/DC) targets.

2.2. **Reactis Simulator**

Reactis Simulator enables users to visualize model execution. Simulator's user interface is similar to those of traditional debuggers from programming languages: it allows users to step through the execution of models by hand as well as set break points. Simulator also supports reverse execution, the replay of tests generated by Reactis Tester, the graphical display of different coverage metrics, and the capability to fine-tune Tester-generated test suites.

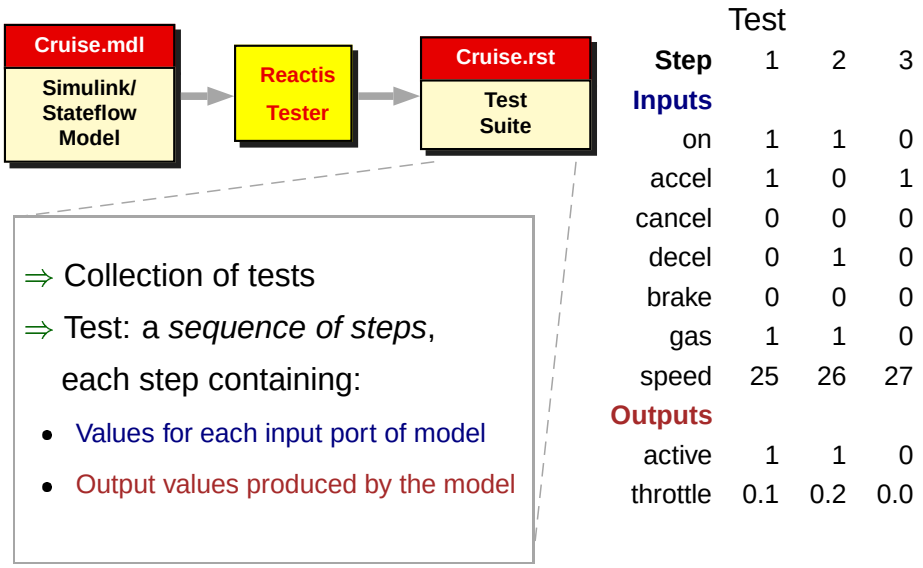

Figure 3: Structure of Tester-generated test suites.

2.3. Reactis Validator

Reactis Validator performs automated searches of models for violations of user-specified requirements. If Validator finds an violation, it returns a test that leads to the problem. This test may then be executed in Reactis Simulator to gain an understanding of the sequence of events that leads to the problem. Validator enables the early detection of design errors and inconsistencies and reduces the effort required for design reviews. Some checks that may be performed with Validator include the following.

- Will a model-variable's value ever fall outside a desired range?
- Will a car's cruise-control maintain vehicle speed within acceptable limits of the set speed?
- Will a plane's thrust reversers ever engage when the aircraft is airborne?
- Will an x-ray machine ever deliver a dangerous dose of radiation?
- Will a cellular phone "hang" when moved from a non-serviced into a serviced area?

3. Advanced Model Validation

A primary benefit of model-based development is that it allows the detection and correction of system-design defects at design (i.e modeling) time, when they are much less expensive and time-consuming to correct, rather than at system-implementation and testing time. Moreover, with proper tool support, the probability of detecting defects at the model level can be significantly increased. In this section, we elaborate on the advanced model-validation capabilities of Reactis that help engineers build better models.

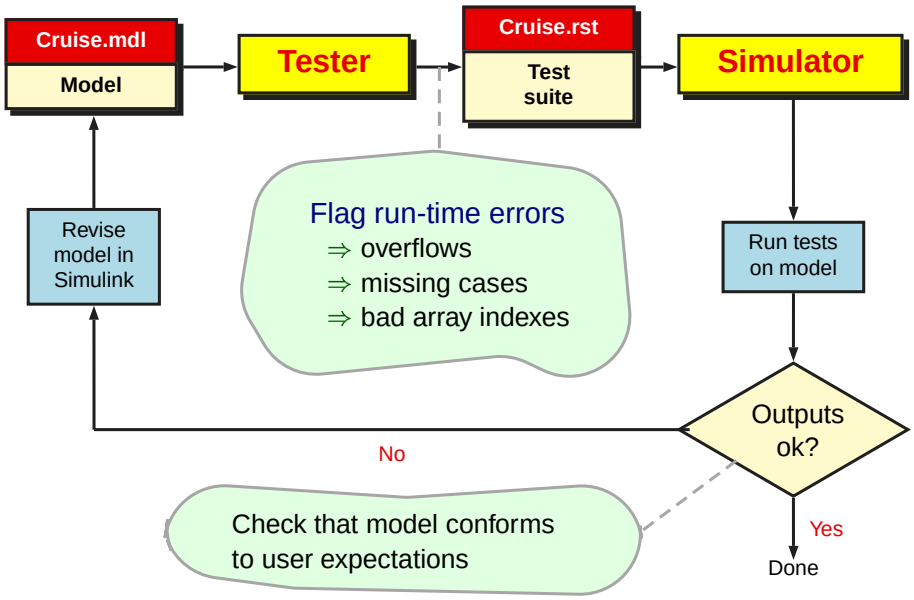

Figure 4: Model debugging with Tester and Simulator.

3.1. Debugging with Tester and Simulator

Reactis Tester and Simulator support model debugging through the automatic generation of test suites that thoroughly exercise the model under investigation (Reactis Tester), and through the visualization of tests as they are executed within the model (Reactis Simulator). One such usage scenario of Tester and Simulator is shown in Figure 4.

Since Tester's guided-simulation test-generation algorithm thoroughly simulates a model during test generation, it often uncovers many run-time errors. For example, overflows, missing cases, and bad array indexes can be discovered. Note that this type of error is also detected when running simulations in Simulink; however, since Tester's guided-simulation engine systematically exercises the model much more thoroughly than random simulation can, the probability of finding such modeling problems is much higher using Reactis.

Tester-generated tests may be executed in Simulator, which offers a number of useful model debugging features; some of these are illustrated in Figure 5. The figure includes a screenshot of Reactis invoked on a Simulink/Stateflow model of an automotive cruise control system. This example is one of several example applications included with the Reactis distribution. The upper window in the figure depicts the model hierarchy on the left and an execution snapshot of a Stateflow diagram from the model on the right.

Reactis allows the user to choose between three distinct sources of input values when visualizing model execution:

1. Input values may be read from a Tester-generated test.
2. They may be generated randomly.
3. They may be supplied interactively by the user.

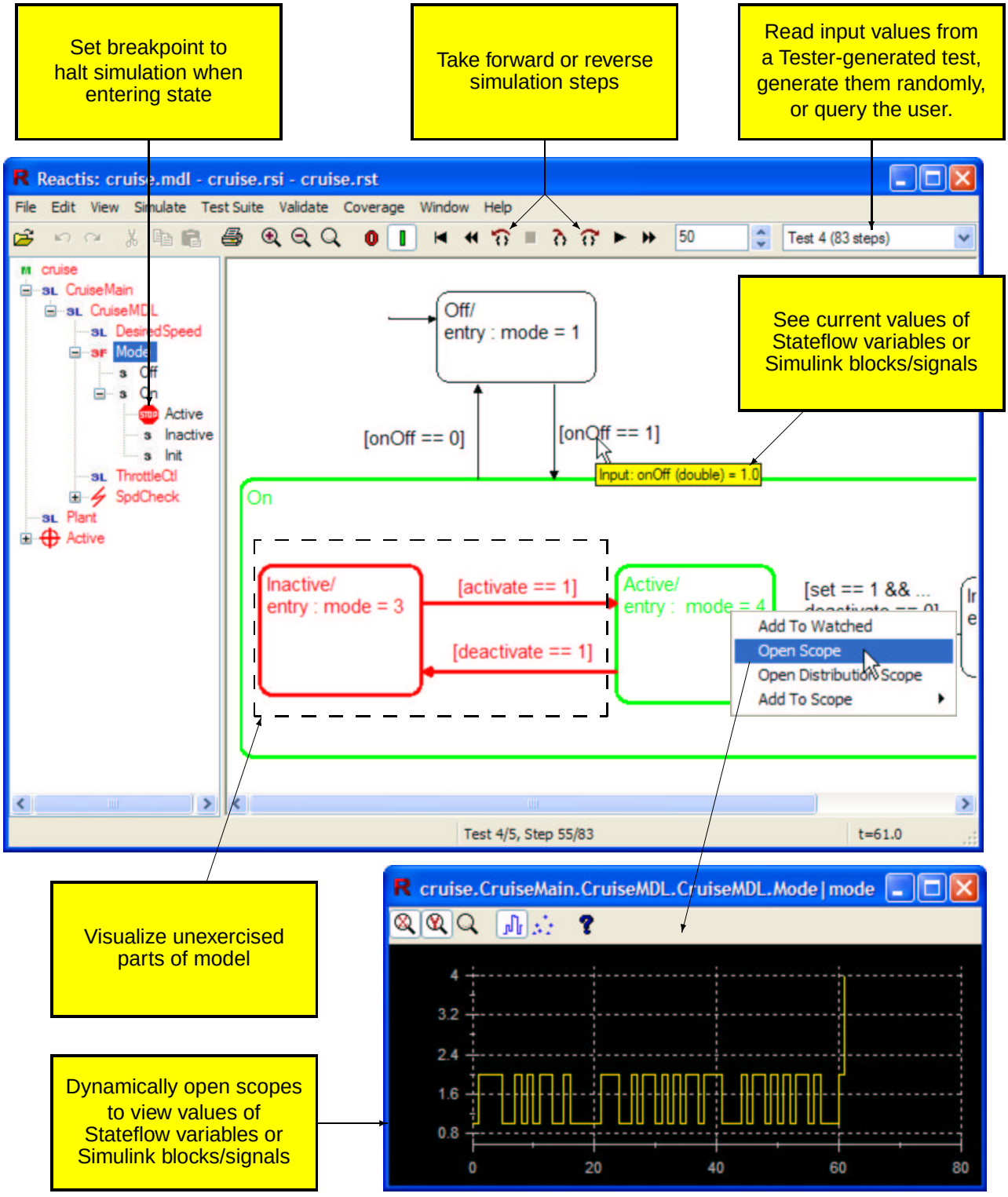

Figure 5: Model-debugging features of Simulator.

As depicted, input values come from Test 4 of a Tester-generated test suite. The other model-debugging facilities illustrated in the figure are as follows.

1. Users may take forward or reverse execution steps when simulating model behavior.

2. Users may dynamically open "scopes" to view the values of Stateflow variables or Simulink blocks and signals. An example scope, depicting how the value of Stateflow variable *mode* varies over time, is shown in the window in the bottom-right corner of the figure. This scope was opened by right-clicking on the mode variable in the diagram panel and selecting "Open Scope."

3. Users may set execution breakpoints. In the example, a breakpoint has been set in state `Active` of the `Mode` subsystem. Therefore, model execution will be suspended when control reaches this state during simulation, allowing the user to carefully examine the model before continuing simulation. Users may query the value of any Simulink block or signal or Stateflow variable by hovering over it. Simulation may be resumed in any input mode, i.e. reading inputs from the test, generating them randomly, or querying the user for them.

4. As shown in the execution snapshot, the current simulation state of the model is highlighted in green and portions of the model that have not yet been exercised during simulation are highlighted in red for easy recognition.

3.2. Validating Properties with Validator

The advanced model-validation capabilities of Reactis are implemented in Reactis Validator. Validator searches for defects and inconsistencies in models. The tool enables users to formulate a requirement as an assertion, attach the assertion to a model, and perform an automated search for a simulation of the model that leads to a violation of the assertion. If Validator finds an assertion violation, it returns a test that leads to the problem. This test may then be executed in Reactis Simulator to gain an understanding of the sequence of events that leads to the problem. Validator also offers an alternative usage under which the tool searches for tests that exercise user-defined coverage targets. The tool enables the early detection of design errors and inconsistencies and reduces the effort required for design reviews.

Figure 6 shows how engineers use Validator. First, a model is instrumented with assertions to be checked and user-defined coverage targets. In the following discussion we shall refer to such assertions and coverage targets as *Validator objectives*. The tool is then invoked on the instrumented model to search for assertion violations and paths leading to the specified coverage targets. The output of a Validator run is a test suite that includes tests leading to objectives found during the analysis.

Validator objectives may be added to any Simulink system or Stateflow diagram in a model. Two mechanisms for formulating objectives in Simulink models are supported:

Expression objectives are C-like boolean expressions.

Diagram objectives are references to subsystems in Simulink / Stateflow libraries.

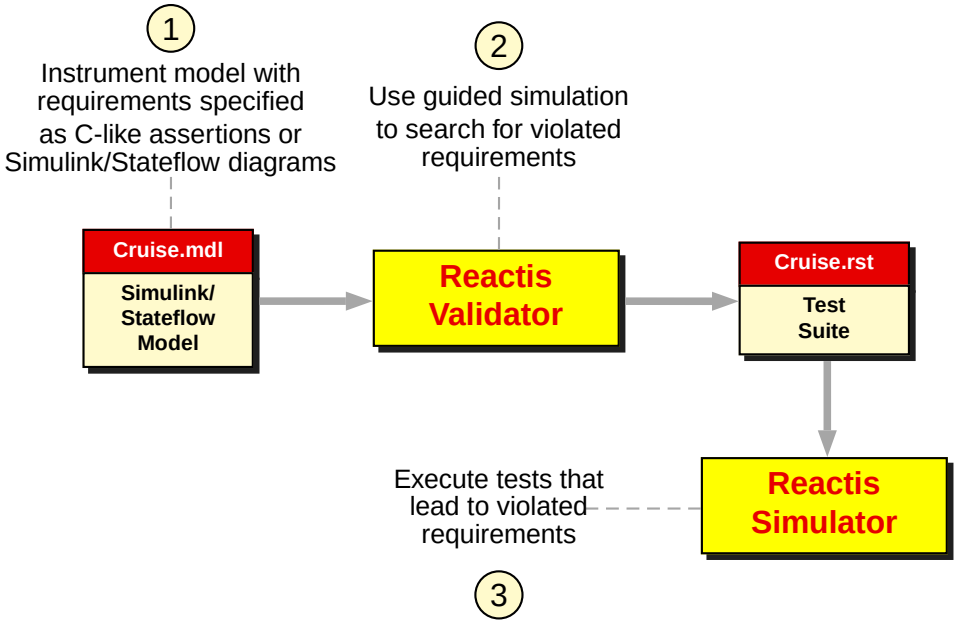

Figure 6: Reactis Validator usage.

Diagram objectives are attached to a model using the Reactis GUI to specify a Simulink system from a library and "wire" it into the model. The diagrams are created using Simulink and Stateflow in the same way standard models are built. After adding a diagram objective to a model, the diagram will be included in the model's hierarchy tree, just as library links are in a model.

Because of its sophisticated model-debugging capabilities, the Reactis tool suite provides significant added value to the market-leading MathWorks Simulink/Stateflow modeling environment. The great virtue of model-level debugging is that it enables engineers to debug a software design before any source code is written. The earlier logic errors are detected, the less costly they are to fix.

4. Enhanced Implementation Testing

The benefits of model debugging and validation have been discussed above. A question that immediately presents itself is: How can the effort expended on these activities be "reused" to support the testing of system implementations? This is the question addressed in this section.

4.1. Software Testing

A crucial aspect of the tests generated by Reactis Tester is that they also store model outputs. Therefore, these tests encode all the information needed to ensure that model-derived source code conforms to its model. Reactis-driven source-code testing proceeds as follows:

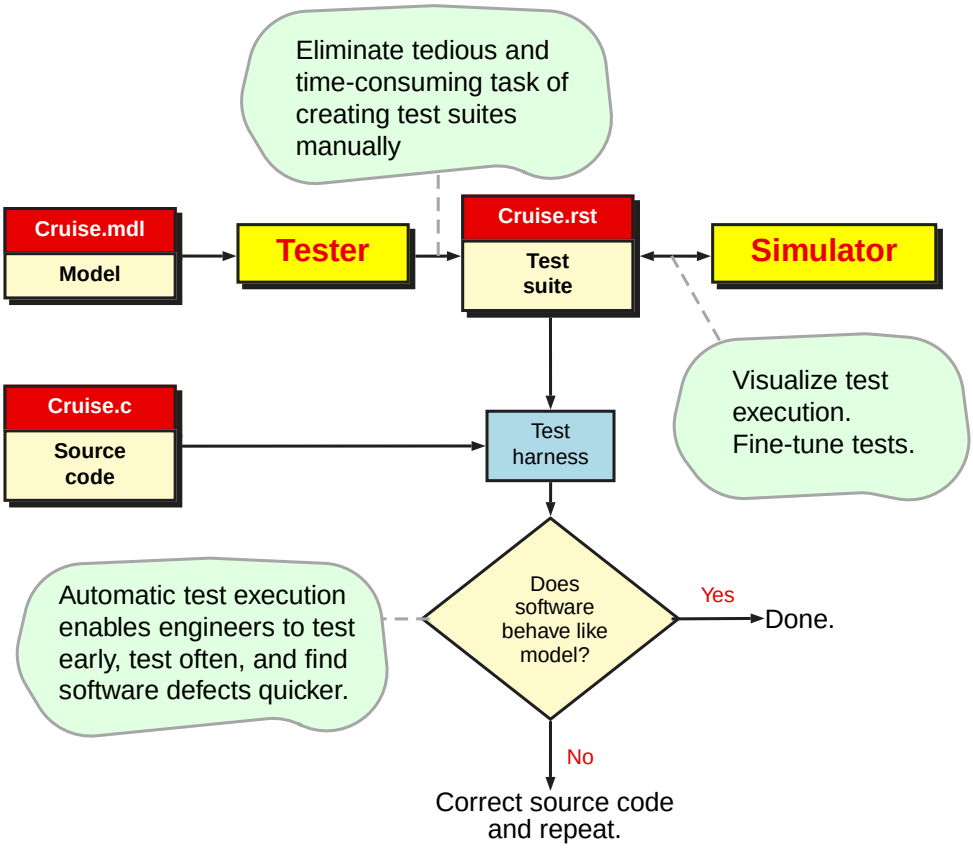

Figure 7: Source-code testing with Reactis.

1. For each test in the suite, execute the software using the input values contained in the test.

2. Compare the output values produced by the software with those stored in the test.

3. Record any discrepancies.

This methodology is referred to as *model-based software testing*, and its key advantage is that the model serves as an "oracle" for testing purposes: the outputs produced by the model can be used as a basis for assessing those generated by the software. If the software does not agree with the model, then the developer can assume that the problem lies within the source code.

The net effect of model-based testing with Reactis is better-quality software at a lower cost. Because good test data is generated and run automatically, less engineer time is required to create and run tests. Because the tests are thorough, the probability of finding bugs is maximized. Because the test suites are compact they may be run quickly. In sum, Reactis dramatically reduces the costs of testing embedded control software.

Figure 7 illustrates how the Reactis tool suite can provide advanced model-based testing of source code. As the figure indicates, the model-based testing protocol supported by Reactis is as follows:

1. The developer provides as input to Reactis a `.mdl` file representing the validated Simulink / Stateflow model of the system under development.

2. Reactis Tester is used to automatically generate a test suite that thoroughly exercises the given model according to the coverage criteria desired by the user.

3. The developer may deploy Reactis Simulator to visualize test execution and to fine tune the tests in the test suite to further improve model coverage.

4. The test suite and the software implementing the model are fed as inputs into a test harness to automate the source-code testing process.

5. By comparing the outputs produced by the software and the model when executing tests in the test suite, deviations in the behavior of the source code from the model are readily detectable and guide the developer in ensuring that the source code conforms to the model.

6. Testing concludes when the source code passes all the tests in the test suite.

4.2. System Testing

After software testing, the next step in certifying a system is to compile the code and test the resulting executable on the platform(s), including microprocessor and associated system software, on which it will eventually be deployed. Such testing is often referred to as system, or integration, testing.

System testing typically involves the use of hardware-in-the-loop (HIL) simulation tools. These HIL tools are expensive, and thus using them as efficiently as possible can promote significant cost savings.

Provided that system-level models are given in Simulink / Stateflow, Reactis can greatly facilitate system testing. As in the case of software testing, test engineers can use Reactis Tester and Reactis Simulator to generate thorough yet compact test suites from these models and feed the test data into their HIL environments in order to check system behavior against model behavior. The compactness of Reactis-generated tests means that expensive HIL hardware need not be "tied up" with long test runs in order to get precise insights into system behavior.

How the Reactis-generated test data may be used in HIL testing will in general depend on the HIL environment used. HIL tools typically provide a scripting facility for defining test runs. Saving Reactis-generated test data in files and then writing scripts to read these files would represent a natural way to inject Reactis test data into these environments.

5. Reverse Engineering Models from Legacy Code

MDB technology can also play an important role with regard to legacy systems. Such systems are often poorly documented and very difficult to modify to meet evolving system requirements

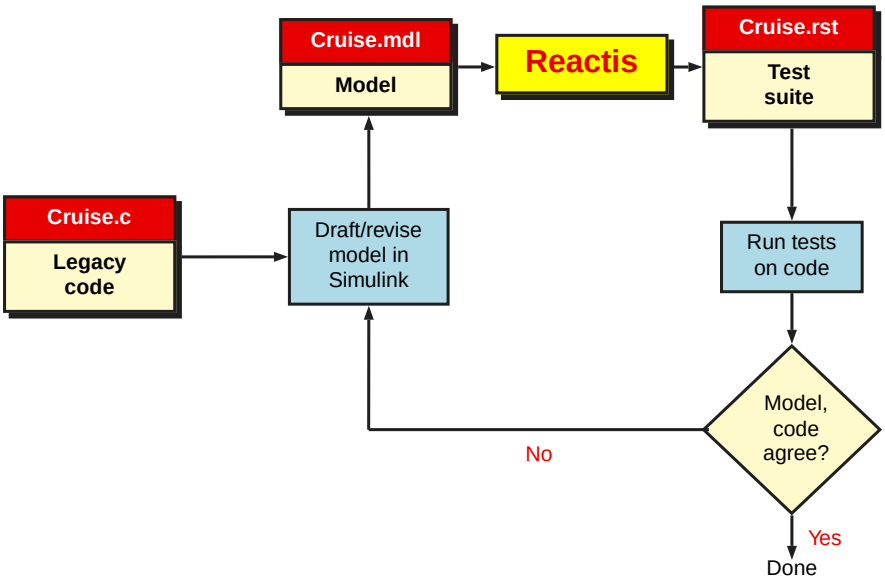

Figure 8: Reverse engineering models with Reactis.

due to the fragile nature of the underlying code. It would benefit developers to have a precise and unambiguous model of the behavior of a legacy system for which they were responsible. Such a model would serve as a formal and executable specification of the legacy system, thereby facilitating system maintenance, documentation, and evolution. The focus of this section is on how Reactis can indeed be used to derive, or "reverse engineer," models from code.

Figure 8 illustrates the process one would follow in order to use Reactis to reverse engineer models from legacy code. Reverse engineering proceeds as follows.

1. The Simulink / Stateflow modeling environment is used to draft a model of the legacy code.

2. The resulting `.mdl` file is fed into Reactis Tester which then automatically generates a test suite from the model. The result is a `.rst` file (a Reactis test-suite file).

3. The generated test suite thoroughly exercises the draft model according to user-selected coverage criteria. Example coverage criteria include Stateflow state and transition coverage and Simulink block coverage. Tester also eliminates redundancy in the test suite it generates in order to eliminate useless test data.

4. Reactis Simulator may then be used to visualize the execution of the tests produced by Reactis Tester, and also to fine-tune the test suite to further improve model coverage.

5. The test suite (the `.rst` file) and the legacy system source code are fed as inputs into a test harness to automate the process of applying the test suite to the legacy code.

6. By comparing the outputs produced by the software and the model on the tests in the test suite, deviations in the behavior of the model from the legacy code are readily detectable and

can be used to guide the user in refining the model to ensure that it faithfully captures the behavior of the legacy system.

7. Reverse engineering of the model concludes when the code passes all tests generated from the model.

The beauty of having a model to go along with a legacy system is that the model serves as a formal and executable specification of the code, thereby easing the tasks of code maintenance, documentation, and evolution.

6. Conclusions

In this paper we have described several ways that the Reactis tool suite provides significant added value to the market-leading MathWorks Simulink/Stateflow modeling environment. Through its sophisticated model-debugging capabilities, Reactis enables engineers to debug a software design before system implementation is undertaken. The earlier design errors are detected, the less costly they are to fix, so better model debugging can reduce overall software costs. We also discussed how the comprehensive yet compact test suites produced by Reactis can dramatically reduce the costs of checking for conformance between a model and system and of reverse-engineering a model from existing control software. By automating tasks that currently require significant manual effort, Reactis cuts development costs. By enabling more thorough testing and validation to be undertaken, it also enables errors to be detected and fixed before systems are fielded and therefore cuts recall and liability costs.

Reactis is available now from Reactive Systems, Inc. Please see the Company's web site at `www.reactive-systems.com` for ordering information and for instructions on how to receive a free 30-day evaluation copy of the software.

References

[Bec00] P. Bechberger. Model-based software development for electronic control units (ECUs). *ATZ/MTZ Special Issue on Automotive Electronics*, February 2000.

[Swo98] R. C. Swortzel. Reducing cycle time and costs of embedded control software using rapid prototyping and automated code generation and test tools. In *Proceedings of International Off-Highway & Powerplant Congress & Exposition*, September 1998.